

8.7 Worst-Case Running Time Analysis

In Section 8.3, we saw how to use asymptotic notation to characterize the *rate of growth* of the number of “basic operations” as a way of analyzing the running time of an algorithm. This approach allows us to ignore details of the computing environment in which the algorithm is run, and machine- and language-dependent implementations of primitive operations, and instead characterize the relationship between the input size and number of basic operations performed.

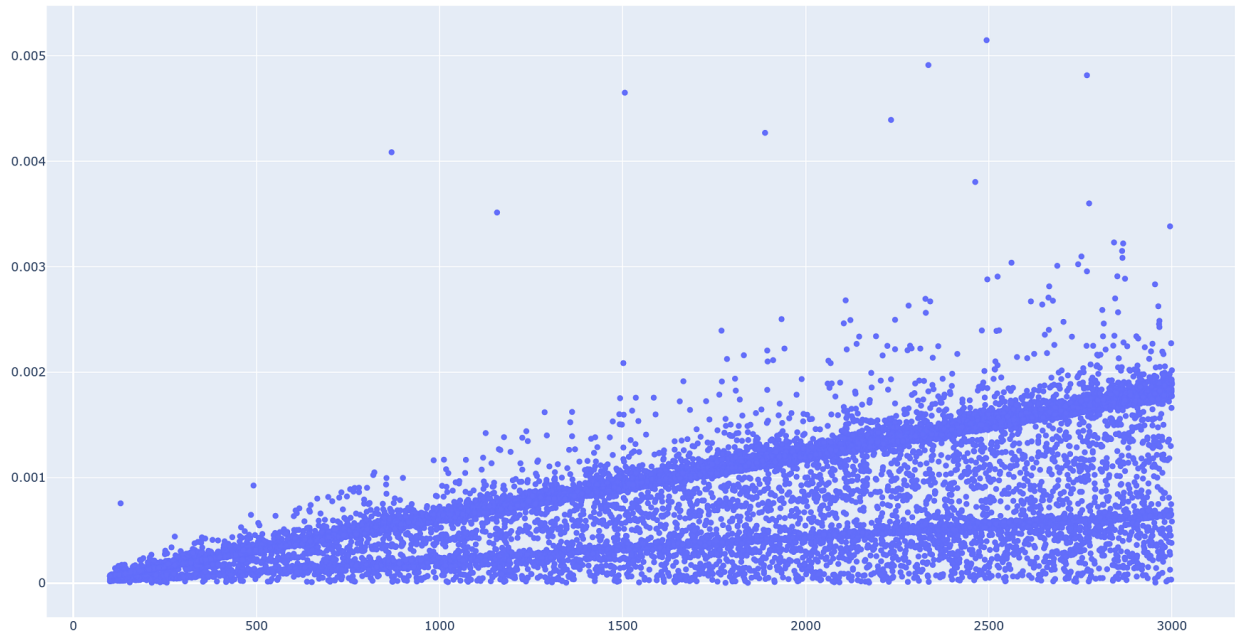
However, this focus on just the input size is a little too restrictive. Even though we can define input size differently for each algorithm we analyze, we tend not to stray too far from the “natural” definitions (e.g., length of list). In practice, though, algorithms often depend on the actual value of the input, not just its size. For example, consider the following function, which searches for an even number in a list of integers.¹

¹ This is very similar to how the `in` operator is implemented for Python lists.

```
def has_even(numbers: list[int]) -> bool:
    """Return whether numbers contains an even element."""
    for number in numbers:
        if number % 2 == 0:
            return True
    return False
```

Because this function returns as soon as it finds an even number in the list, its running time is not necessarily proportional to the length of the input list.

The running time of a function can vary even when the input size is fixed. Or using the notation we learned earlier this chapter, the inputs in $\mathcal{I}_{has_even,10}$ do *not* all have the same runtime. The question “what is *the* running time of `has_even` on an input of length n ?” does not make sense, as for a given input the runtime depends not just on its length but on which of its elements are even. We illustrate in the following plot, which shows the results of using `timeit` to measure the running time of `has_evens` on randomly-chosen lists. While every timing experiment has some inherent uncertainty in the results, the spread of running times cannot be explained by that alone!



Because our asymptotic notation is used to describe the growth rate of *functions*, we cannot use it to describe the growth of a whole range of values with respect to increasing input sizes. A natural approach to fix this problem is to focus on the *maximum* of this range, which corresponds to the *slowest* the algorithm could run for a given input size.

Definition. Let `func` be a program. We define the function $WC_{func} : \mathbb{N} \rightarrow \mathbb{N}$, called the **worst-case running time function of `func`**, as follows:²

² Here, “running time” is measured in exact number of basic operations. We are taking the maximum of a set of numbers, *not* a set of asymptotic expressions.

$$WC_{func}(n) = \max \{ \text{running time of executing } func(x) \mid x \in \mathcal{I}_{func,n} \}$$

Note that WC_{func} is a function, not a (constant) number: it returns the maximum possible running time for an input of size n , for every natural number n . And because it is a function, we can use asymptotic notation to describe it, saying things like “the worst-case running time of this function is $\Theta(n^2)$.”

The goal of a *worst-case runtime analysis* for `func` is to find an elementary function f such that $WC_{func} \in \Theta(f)$.

However, it takes a bit more work to obtain tight bounds on a worst-case running time than on the runtime functions of the previous section. It is difficult to compute the *exact maximum* number of basic operations performed by this algorithm for every input size, which requires that we identify an input for each input size, count its maximum number of basic operations, and then prove that every input of this size takes at most this number of operations. Instead, we will generally take a two-pronged approach: proving matching *upper* and *lower bounds* on the worst-case running time of our algorithm.

Upper bounds on the worst-case runtime

Definition. Let `func` be a program, and WC_{func} its worst-case runtime function. We say that a function $f : \mathbb{N} \rightarrow \mathbb{R}^{\geq 0}$ is an **upper bound on the worst-case runtime** when $WC_{func} \in \mathcal{O}(f)$.

To get some intuition about what an upper bound on the worst-case running means, suppose we use absolute dominance rather than Big-O. In this case, there's a very intuitive way to expand the phrase " WC_{func} is absolutely dominated by f ":

$$\begin{aligned} & \forall n \in \mathbb{N}, WC_{func}(n) \leq f(n) \\ \iff & \forall n \in \mathbb{N}, \max \{ \text{running time of executing } func(x) \mid x \in \mathcal{I}_{func,n} \} \leq f(n) \\ \iff & \forall n \in \mathbb{N}, \forall x \in \mathcal{I}_{func,n}, \text{ running time of executing } func(x) \leq f(n) \end{aligned}$$

The last line comes from the fact that if we know the maximum of a set of numbers is less than some value K , then *all* numbers in that set must be less than K . Thus an upper bound on the worst-case runtime is equivalent to an upper bound on the runtimes of *all* inputs.

Now when we apply the definition of Big-O instead of absolute dominance, we get the following translation of $WC_{func} \in \mathcal{O}(f)$:

$$\exists c, n_0 \in \mathbb{R}^+, \forall n \in \mathbb{N}, n \geq n_0 \Rightarrow (\forall x \in \mathcal{I}_{func,n}, \text{ running time of executing } func(x) \leq c \cdot f(n))$$

To approach an analysis of an upper bound on the worst-case, we typically find a function g such that WC_{func} is absolutely dominated by g , and then find a simple function f such that $g \in \mathcal{O}(f)$. But how do we find such a g ? And what does it mean to upper bound *all* runtimes of a given input size? We'll illustrate the technique in our next example.

Example. Find an asymptotic upper bound on the worst-case running time of `has_even`.

Discussion. The intuitive translation using absolute dominance is usually enough for an upper bound analysis. In particular, the $\forall n \in \mathbb{N}, \forall x \in \mathcal{I}_{func,n}$ begins with two universal quantifiers, and just knowing this alone should anticipate how we'll start our proof, using the same techniques of proof we learned earlier!

Running time analysis. (Upper bound on worst-case)

First, let $n \in \mathbb{N}$ and let `numbers` be an *arbitrary* list of length n .

Now we'll analyse the running time of `has_even`, except we *can't* assume anything about the values inside `numbers`, because it's an arbitrary list. But

we can still find an *upper bound* on the running time:

- The loop (for number in numbers) iterates *at most* n times. Each loop iteration counts as a single step (because it is constant time), so the loop takes *at most* $n \cdot 1 = n$ steps in total.
- The return False statement (if it is executed) counts as 1 basic operation.

Therefore the running time is *at most* $n + 1$, and $n + 1 \in \mathcal{O}(n)$. So we can conclude that the worst-case running time of has_even is $\mathcal{O}(n)$. ■

Note that we did *not* prove that has_even(numbers) takes exactly $n + 1$ basic operations for an arbitrary input numbers (this is false); we only proved an *upper bound* on the number of operations. And in fact, we don't even care that much about the exact number: what we ultimately care about is the asymptotic growth rate, which is linear for $n + 1$. This allowed us to conclude that the worst-case running time of has_even is $\mathcal{O}(n)$.

But because we calculated an upper bound rather than an exact number of steps, we can only conclude a Big-O, not Theta bound: we don't yet know that this upper bound is tight.³

³ If this is surprising, note that we could have done the above proof but replaced $n + 1$ by $5000n + 110$ and it would still have been mathematically valid.

Lower bounds on the worst-case runtime

So how do we prove our upper bound is tight? Since we've just shown that $WC_{has_even}(n) \in \mathcal{O}(n)$, we need to prove the corresponding lower bound $WC_{has_even}(n) \in \Omega(n)$. But what does it mean to prove a lower bound on the maximum of a set of numbers? Suppose we have a set of numbers S , and say that "the maximum of S is at least 50." This doesn't tell us what the maximum of S actually is, but it does give us one piece of information: there has to be a number in S which is at least 50.

The key insight is that the converse is also true—if I tell you that S contains the number 50, then you can conclude that the maximum of S is at least 50.

$$\max(S) \geq 50 \Leftrightarrow (\exists x \in S, x \geq 50)$$

Using this idea, we'll give a formal definition for a lower bound on the worst-case runtime of an algorithm.

Definition. Let func be a program, and WC_{func} is worst-case runtime function. We say that a function $f : \mathbb{N} \rightarrow \mathbb{R}^{\geq 0}$ is a **lower bound on the worst-case runtime** when $WC_{func} \in \Omega(f)$.

In an analogous fashion to the upper bound, we unpack this definition first by using absolute dominance:

$$\begin{aligned} & \forall n \in \mathbb{N}, WC_{func}(n) \geq f(n) \\ \iff & \forall n \in \mathbb{N}, \max \{ \text{running time of executing } func(x) \mid x \in \mathcal{I}_{func,n} \} \geq f(n) \\ \iff & \forall n \in \mathbb{N}, \exists x \in \mathcal{I}_{func,n}, \text{ running time of executing } func(x) \geq f(n) \end{aligned}$$

And then using Omega:

$$\exists c, n_0 \in \mathbb{R}^+, \forall n \in \mathbb{N}, n \geq n_0 \Rightarrow (\exists x \in \mathcal{I}_{func,n}, \text{ running time of executing } func(x) \geq c \cdot f(n))$$

Remarkably, the crucial difference between this definition and the one for upper bounds is a change of quantifier: now the input x is existentially quantified, meaning we get to pick it. Or really, our goal is to find an **input family**—a *set* of inputs, one per input size n —whose runtime is asymptotically larger than our target lower bound.

For example, in `has_even` we want to prove that the worst-case running time is $\Omega(n)$ to match the $\mathcal{O}(n)$ upper bound, and so we want to find an input family where the number of steps taken is $\Omega(n)$. Let's do that now.

Example. Find an asymptotic lower bound on the worst-case running time of `has_even`.

Discussion. Again, we'll just remind you of the quantifiers from the intuitive "absolute dominance" version of the lower bound definition: $\forall n \in \mathbb{N}, \exists x \in \mathcal{I}_n$. This will inform how we start our proof.

Running time analysis. (Lower bound on worst-case)

Let $n \in \mathbb{N}$. Let `numbers` be the list of length n consisting of all 1's. Now we'll analyse the (exact) running time of `has_even` on this input.

In this case, the `if` condition in the loop is always false, so the loop never stops early. Therefore it iterates exactly n times (once per item in the list), with each iteration taking one step.

Finally, the `return False` statement executes, which is one step. So the total number of steps for this input is $n + 1$, which is $\Omega(n)$. ■

Putting it all together

Finally, we can combine our upper and lower bounds on WC_{has_even} to obtain a tight asymptotic bound.

Example. Find a *tight* bound on the worst-case running time of `has_even`.

Running time analysis. Since we've proved that WC_{has_even} is $\mathcal{O}(n)$ and $\Omega(n)$, it is $\Theta(n)$. ■

To summarize, to obtain a tight bound on the worst-case running time of a function, we need to do two things:

- Use the properties of the code to obtain an *asymptotic upper bound* on the worst-case running time. We would say something like $WC_{func} \in \mathcal{O}(f)$.
- Find a family of inputs whose running time is $\Omega(f)$.⁴ This will prove that

⁴ Almost always we find an input family whose running time is $\Theta(f)$, but strictly speaking only $\Omega(f)$ is required.

$$WC_{func} \in \Omega(f).$$

- After showing that $WC_{func} \in \mathcal{O}(f)$ and $WC_{func} \in \Omega(f)$, we can conclude that $WC_f \in \Theta(f)$.

A note about best-case runtime

In this section, we focused on worst-case runtime, the result of taking the *maximum* runtime for every input size. It is also possible to define a best-case runtime function by taking the minimum possible runtimes, and obtain tight bounds on the best case through an analysis that is completely analogous to the one we just performed. In practice, however, the best-case runtime of an algorithm is usually not as useful to know—we care far more about knowing just how *slow* an algorithm is than how fast it can be.

Early returning in Python built-ins

We've encountered a few different Python functions and methods whose running time depends on more than just the size of their inputs. We alluded to one at the start of this chapter: the list search operation using the keyword `in`:

```
>>> lst = list(range(0, 1000000))
>>> timeit.timeit('0 in lst', number=10, globals=globals())
8.299997716676444e-06
>>> timeit.timeit('-1 in lst', number=10, globals=globals())
0.17990550000104122
```

In the first `timeit` expression, `0` appears as the first element of `lst`, and so is found immediately when the search occurs. In the second, `-1` does not appear in `lst` at all, and so all one-million elements of `lst` must be checked, resulting in a running-time that is proportional to the length of the list. *The worst-case running time of the `in` operation for lists is $\Theta(n)$, where n is the length of the list.*

We have also seen two more functions that are implemented using an early return: `any` and `all`. Because `any` searches for a single `True` in a collection, it stops the first time it finds one. Similarly, because `all` requires that all elements of a collection be `True`, it stops the first time it finds a `False` value.

```
>>> all_trues = [True] * 1000000
>>> all_falses = [False] * 1000000
>>> timeit.timeit('any(all_trues)', number=10, globals=globals())
8.600000001024455e-06
>>> timeit.timeit('any(all_falses)', number=10, globals=globals())
0.10643419999905745
>>> timeit.timeit('all(all_trues)', number=10, globals=globals())
0.10217570000168053
>>> timeit.timeit('all(all_falses)', number=10, globals=globals())
6.300000677583739e-06
```

So in the above example:

- `any(all_trues)` returns `True` immediately after checking the first list element.
- `any(all_falses)` returns `False` only after checking all one-million list elements.
- `all(all_trues)` returns `True` only after checking all one-million list elements.
- `all(all_falses)` returns `False` immediately after checking the first list element.

So `any` and `all` have a worst-case running time of $\Theta(n)$, where n is the size of the input collection. But in practice they can be much faster if they encounter the “right” boolean value early on!

any, all, and comprehensions

There is one subtlety that often catches students by surprise when they attempt to call `any/all` on a comprehension and expect a quick result. Let’s see a simple example:

```
>>> timeit.timeit('any([x == 0 for x in range(0, 1000000)])', number=10,
                  globals=globals())
0.7032962000012049
```

That’s a lot slower than we would expect, given that the first element checked is `x = 0`! The result is similar if we try to use a set comprehension instead of a list comprehension:

```
>>> timeit.timeit('any({x == 0 for x in range(0, 1000000)})', number=10,
                  globals=globals())
0.6538308000017423
```

The subtlety here is that in both cases, *the full comprehension is evaluated before any is called*. As we discussed in 8.5 Analyzing Comprehensions and While Loops, the running time of evaluating a comprehension is proportional to the size of the source collection of the comprehension—in our example, that’s `range(0, 1000000)`, which contains one-million numbers.

But all is not lost! In practice, Python programmers *do* use `any/all` with comprehensions, but they do so by writing the comprehension expression in the function call without any surrounding square brackets or curly braces:

```
>>> any(x == 0 for x in range(0, 1000000))
True
```

This is called a **generator comprehension**, and is used to produce a special Python collection data type called a **generator**. We won’t use generators or generator comprehensions very much at all in this course, but what we want you to know about them here is that unlike set/list comprehensions, generator comprehensions do not evaluate their elements all at once, but instead only when they are needed by the function being called. This means that our above `any` call achieves the fast running time we initially expected:

```
>>> timeit.timeit('any(x == 0 for x in range(0, 1000000))', number=10,
                  globals=globals())
4.050000279676169e-05
```

Now, only the `x = 0` value from the generator comprehension gets evaluated; none of the other possible values (`x = 1, 2, ..., 999999`) are ever checked by the `any` call!

Don’t assume bounds are tight!

It is likely unsatisfying to hear that upper and lower bounds really are distinct things that must be computed separately. Our intuition here pulls us towards the bounds being “obviously” the same, but this is really a side effect of the examples we have studied so far in this course being rather straightforward. But this won’t always be the case: the study of more complex algorithms and data structures exhibits quite a few situations where obtaining an upper bound on the running time involves a completely different analysis than the lower bound.

Let’s look at one such example that deals with manipulating strings.

Example. We say that a string is a **palindrome** when it can be read the same forwards and backwards; example of palindromes are “abba”, “racecar”, and “z”.⁵ We say that a string s_1 is a

⁵ Every string of length 1 is a palindrome.
--

prefix of another string s_2 when s_1 is a substring of s_2 that starts at index 0 of s_2 . For example, the string “abc” is a prefix of “abcdef”.

The algorithm below takes a non-empty string as input, and returns the length of the longest prefix of that string that is a palindrome. For example, the string “attack” has two non-empty prefixes that are palindromes, “a” and “atta”, and so our algorithm will return 4.

```
def palindrome_prefix(s: str) -> int:
    n = len(s)
    for prefix_length in range(n, 0, -1): # goes from n down to 1
        # Check whether s[0:prefix_length] is a palindrome
        is_palindrome = all(s[i] == s[prefix_length - 1 - i]
                             for i in range(0, prefix_length))

        # If a palindrome prefix is found, return the current length.
        if is_palindrome:
            return prefix_length
```

There are a few interesting details to note about this algorithm:

- The for loop iterable is `range(n, 0, -1)`—the third argument `-1` causes the loop variable to *start* at `n` and decrease by 1 at each iteration. In other words, this loop is checking the possible prefixes starting with the longest prefix (length `n`) and working its way to the shortest prefix (length 1).
- The call to `all` checks pairs of characters starting at either end of the current prefix. It uses a *generator comprehension* (like we discussed above) so that it can stop early as soon as it encounters a mismatch (i.e., when `s[i] != s[prefix_length - 1 - i]`).

- Even though the only return statement is inside the for loop, this algorithm is guaranteed to find a palindrome prefix, since the first letter of s by itself is a palindrome.

The code presented here is structurally simple. Indeed, it is not too hard to show that the worst-case runtime of this function is $\mathcal{O}(n^2)$, where n is the length of the input string. What is harder, however, is showing that the worst-case runtime is $\Omega(n^2)$. To do so, we must find an input family whose runtime is $\Omega(n^2)$. There are two points in the code that can lead to fewer than the maximum loop iterations occurring, and we want to find an input family that avoids both of these.

The difficulty is that these two points are caused by different types of inputs! The call to `all` can stop as soon as the algorithm detects that a prefix is *not* a palindrome, while the return statement occurs when the algorithm has determined that a prefix *is* a palindrome! To make this tension more explicit, let's consider two extreme input families that seem plausible at first glance, but which do not have a runtime that is $\Omega(n^2)$.

- The entire string s is a palindrome of length n . In this case, in the first iteration of the loop, the entire string is checked. The `all` call checks all pairs of characters, but unfortunately this means that `is_palindrome = True`, and the loop returns during its very first iteration. Since the `all` call takes n steps, this input family takes $\Theta(n)$ time to run.
- The entire string s consists of n different letters. In this case, the only palindrome prefix is just the first letter of s itself. This means that the loop will run for all n iterations, only returning in its last iteration (when `prefix_length == 1`). However, the `all` call will always stop after just one step, since it starts by comparing the first letter of s with another letter, which is guaranteed to be different by our choice of input family. This again leads to a $\Theta(n)$ running time.

The key idea is that we want to choose an input family that *doesn't* contain a long palindrome (so the loop runs for many iterations), but whose prefixes are close to being palindromes like palindromes (so the `all` call checks many pairs of letters). Let $n \in \mathbb{Z}^+$. We define the input s_n as follows:

- $s_n[\lceil n/2 \rceil] = b$
- Every other character in s_n is equal to a .

For example, $s_4 = aaba$ and $s_{11} = aaaaaabaaa$.

Note that s_n is very close to being a palindrome: if that single character b were changed to an a , then s_n would be the all- a 's string, which is certainly a palindrome. But by making the centre character a b , we not only ensure that the longest palindrome of s_n has length roughly $n/2$ (so the loop iterates roughly $n/2$ times), but also that the “outer” characters of each prefix of s_n containing more than $n/2$ characters are all the same (so the `all` call checks many pairs to find the mismatch between a and b). It turns out that this input family does indeed have an $\Omega(n^2)$ runtime! We'll leave the details as an exercise.